# Determinating Timing Channels in Compute Clouds

Amittai Aviram, Sen Hu, Bryan Ford
Yale University

Ramakrishna Gummadi
University of Massachusetts Amherst



## ABSTRACT

Timing side-channels represent an insidious security challenge for cloud computing, because: (a) massive parallelism in the cloud makes timing channels pervasive and hard to control; (b) timing channels enable one customer to steal information from another without leaving a trail or raising alarms; (c) only the cloud provider can feasibly detect and report such attacks, but the provider's incentives are *not* to; and (d) resource partitioning schemes for timing channel control undermine statistical sharing efficiency, and, with it, the cloud computing business model. We propose a new approach to timing channel control, using *provider-enforced deterministic execution* instead of resource partitioning to eliminate timing channels within a shared cloud domain. Provider-enforced determinism prevents execution timing from affecting the results of a compute task, however large or parallel, ensuring that a task's outputs leak no timing information apart from explicit timing inputs and total compute duration. Experiments with a prototype OS for deterministic cloud computing suggest that such an approach may be practical and efficient. The OS supports deterministic versions of familiar APIs such as processes, threads, shared memory, and file systems, and runs coarse-grained parallel tasks as efficiently and scalably as current timing channel-ridden systems.

## 1. INTRODUCTION

It is hotly debated whether individuals and companies should trust cloud providers with sensitive information, but few would suggest that a cloud customer should trust the provider *and* all the provider's other customers. Yet this may soon be the cloud's *de facto* security model—if it isn't already—due to timing channels.

Timing channels are well-known and well-studied [20,37], originally driven by military-grade security demands. They have gained broader relevance, however, in the context of commercially applicable information flow control [17, 38], and due to the discovery that computations *unintentionally* broadcast sensitive information via numerous timing channels in shared environments. A sensitive computation sharing a CPU core with an attacker, through either time division or hyperthreading, is akin to standing behind a transparent shower door: e.g., an attacker may steal information from the victim via the shared L1 data cache [28], shared functional units [35], the branch target cache [2], or the instruction cache [1].

Most of the above attacks were demonstrated between processes on a conventional OS, but per-customer VMs on a provider-owned machine share resources in essentially the same way, making the results theoretically applicable to clouds—especially those relying on "container-based" virtualization [32]. Timing attacks have even been demonstrated specifically on VMs commonly used in clouds [30], although it is not yet clear how easily these lab-based experiments could be replicated in a noisy commercial cloud.

Whether timing channels represent an immediate security threat or merely a hairline fracture, it is worth repeating the security adage, "attacks never get worse; they only get better." Today's timing-channel exploits pick low-hanging fruit, extracting information from only one high-bandwidth timing channel at a time via straightforward analysis techniques. Shared computing environments have many other timing channels, such as L3 caches shared between cores, memory and I/O busses, and cluster interconnects. There are probably ways to extract weaker signals from stronger noise, aggregate information from low-rate leaks over time, correlate leaks across multiple channels, etc. Attack amplification techniques applicable to arbitrary timing channels have already appeared [29]. It would simply be foolish for us to expect timing attacks *not* to continue getting more effective and more practical over time.

In the rest of this paper, we set aside the "imminence of threat" debate and simply assume that at *some* point, sooner or later, timing channels will become an important cloud security issue. We focus here on understanding the basic nature of the timing channel problem in the cloud context, independent of specific channels and attacks, and on discovering potential solutions compatible with the requirements of cloud environments. We focus in particular on timing channels *internal* to a cloud: other side-channels, such as those derived from a client's communication with a cloud-based service [12], are also important but beyond our present scope.

We make three main contributions. First, we identify four ways the cloud computing model amplifies timing channel security risks compared with traditional infrastructure. Second, we propose a new method of timing channel control based on provider-enforced deterministic execution, which aggregates *all* internal timing channels into a single controllable channel at the cloud's border. Third, we present a proof-of-concept cloud computing OS that enforces determinism, with preliminary results suggesting that it could support parallel cloud applications efficiently without sacrificing the cloud provider's flexibility in allocating resources to clients.

## 2. TIMING CHANNELS IN THE CLOUD

Current cloud privacy discussions focus on the provider's obligation to enforce security and earn the customer's trust. These discussions presuppose the provider's full awareness of the security risks from which it must shield the customer [24,27]. But exposure to malice from another customer's software may be hard for the provider to detect or prevent without careful consideration of the cloud's architecture. Timing channels typify such insidious risks.

Although timing channels represent an important security risk in any shared infrastructure, the cloud model exacerbates these risks in at least four specific ways, which we discuss below. The first two points are well-known to some but worth repeating, while to our knowledge the second two have not previously been discussed.

### *Parallelism creates pervasive timing channels.*

In the days of uniprocessors and single-threaded processes, it was possible to control timing channels by limiting untrusted processes' access to high-resolution clocks and timers, and to other I/O devices that can behave like clocks [20, 37]. But today's increas-

```
volatile long long timer = 0;

void *timer_func(void *)
  { while (1) timer++; }

main() {
  pthread_create(&timer_thread, NULL,
                 timer_func, NULL);
  ...
  // Read the "current time"
  long long timestamp = timer;
  ...
}
```

**Figure 1: Implementing a high-resolution reference clock using threads, when no explicit hardware clocks are available.**

ingly parallelism-oriented hardware—especially in the massively parallel cloud context—creates numerous implicit, high-resolution clocks that have nothing to do with I/O. Hardware caches and interconnects in their many forms all represent shared resources that can be modulated [1, 2, 28, 35]. A thread running in a loop can create a high-resolution reference clock [37], as illustrated by the trivial code in Figure 1, even if the OS or VM has virtualized or disabled all "explicit" hardware clocks. Even processes with no access to explicit clocks, timers, or other devices, can thus use parallelism-derived implicit clocks to exploit timing channels.

*Insider attacks become outsider attacks.*

With notable exceptions [10], timing channel exploits usually require the attacker to run a sophisticated, CPU-intensive program on the victim's machine. On private infrastructure, this usually means the attacker must be an "insider" or have already compromised the machine. But a cloud provider's business is to run any paying customer's computation with "no questions asked." Since the provider may colocate arbitrary customers' computations on a given machine without the knowledge or consent of either, a timing attack exploitable only by "insiders" on private infrastructure may be mounted by malicious "outsiders" in the cloud. An attacker may simply "fish" for secrets without even knowing the identity of the co-resident victim, by monitoring timing channels for SSH keystrokes for example, or the attacker may deliberately attempt to obtain co-residency with a specific target [30].

*Cloud-based timing attacks are unlikely to be caught.*

The owner of private infrastructure has the right to monitor and inspect any running software to detect malicious code. Cloud customers cannot monitor other customers' computations to protect themselves against timing attacks, however (except by engaging in "counter-espionage" attacks themselves), and cloud providers have no prerogative to monitor their customers' computations due to customer privacy concerns. Since a timing attack leaves no trail of compromised protection mechanisms, successful timing attacks are unlikely to raise alarms and will probably just go unnoticed. Thus, providers risk nothing by leaving timing attacks undetected and unreported, whereas monitoring customers in order to detect and report such attacks may invite privacy lawsuits.

*Controlling timing channels via resource partitioning undermines the cloud's elasticity and business model.*

One general approach to controlling timing channels is to limit the rate at which one user's demand for a shared resource may visibly affect the resource's availability to another user, either by statically partitioning the resource or injecting noise into scheduling decisions. Recent cache partitioning proposals exemplify this approach [21]. These methods limit the provider's ability to oversubscribe and statistically multiplex shared hardware efficiently among users, however, undermining the basic business model of cloud computing. Without statistical multiplexing, the cloud loses its elasticity, leaving the provider essentially selling only private infrastructure hosting and outsourced management services.

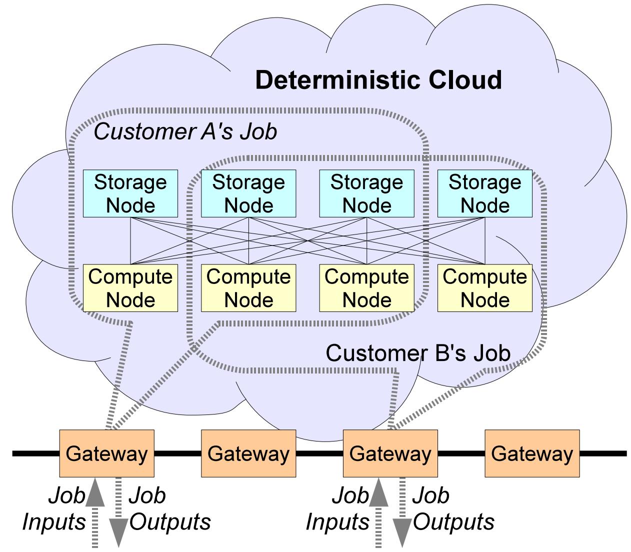


**Figure 2: Timing-hardened cloud architecture. Gateways accept requests, dispatch deterministic jobs into the cloud, then return job results that depend *only* on explicit job inputs, and not on internal timing.**

## 3. A TIMING-HARDENED CLOUD

We now explore a cloud computing architecture that closes all internal timing channels, regardless of number and types of shared resources, leaving only one controllable timing channel at the boundary. The basic idea is to make the cloud behave like a deterministic batch job processor, reminiscent of early mainframes.

A computation needs access to two "clocks" to exploit any timing channel: a *reference clock* and a clock that can be *modulated* [37]. While standard approaches to timing channel control attempt to limit visible clock modulation, our approach is to eliminate all internal reference clocks—even in the presence of parallelism.

### 3.1 Provider-Enforced Determinism

As illustrated in Figure 2, a set of gateway nodes at the cloud's boundary accepts job requests, including any inputs the job requires. Upon completion, the gateway returns the job's outputs, which depend *only* on explicit inputs, and not on timings of operations within the cloud. For each job, the cloud provider effectively computes a *pure mathematical function*, whose outputs depend only on the job's explicit customer-provided inputs, and nothing else. The provider's cloud OS or VMM enforces this determinism, ensuring that even malicious guest code can do nothing to make its results depend on internal timing or other implicit inputs.

To process each job, the provider's gateway breaks the job into smaller work units and uses load-balancing algorithms controlled by the provider to distribute work among cloud servers. These servers may communicate internally while performing a job, pro-

vided communication timing cannot affect computed results.

A customer's job may also read and write the customer's persistent data stored in the cloud, provided any writes remain invisible both externally and to other jobs until the writing job completes. Each job in effect executes within a provider-enforced transaction.

The provider may statistically multiplex different customers' jobs freely onto shared hardware within the cloud, with no static partitioning or scheduling noise injection. Provider-enforced determinism nevertheless ensures that no timing or other nondeterministic information leaks from one guest computation to another, and only one unit of timing information per job leaks to the outside world: namely the total time the job took to complete. This remaining timing channel leaks only heavily aggregated information that is unlikely to be easily exploitable, and the provider can limit this timing channel's information flow rate by returning job results to customers on a periodic schedule—e.g., once per millesecond, second, or minute—rather than immediately on job completion.

## 3.2 Applicability of the Architecture

The applicability of this cloud architecture depends on two questions: whether a strictly deterministic execution environment can provide a practical programming model for cloud applications, and whether such a deterministic environment can be efficient enough. We address the first question here and the second in Section 4.

This architecture may be readily applicable to many large, parallel, compute-bound applications such as scientific computing, rendering, and data analysis. Nondeterminism in parallel applications is usually undesired [9, 23], so eliminating it benefits the developer. The only common *intentional* nondeterminism in such applications is for internal performance optimization purposes—e.g., distributing work items to workers according to dynamic availability and load—and our architecture delegates these functions to the cloud provider. Determinism thus simplifies the customer's programming task by eliminating pervasive heisenbugs [25], making all bugs reproducible [22], and offloading load-balancing responsibilities to the provider. Applicability thus reduces to the efficiency question.

While large compute-bound applications fit the proposed architecture most naturally, more interactive uses may be feasible as well. A deterministic cloud might host interactive web applications, for example, as follows. The provider's gateway nodes act as generic front-end Web servers, accepting HTTP requests from remote clients and converting them into deterministic job submissions on behalf of the web application's owner. The gateway attaches a job creation timestamp to each job's inputs, enabling the application to "tell time" at job granularity. A job's results can request the gateway to start a follow-up job at a future time, enabling the web application to implement timeouts, push notifications on persistent sockets, etc. The remaining questions are whether such a "gateway-driven" web programming model can be made sufficiently familiar for customers implementing web applications, and whether the provider can support job creation and dispatch at sufficiently high rate and fine granularity to handle customer response time requirements. We believe both of these questions can be answered be answered positively, the first using appropriate runtime libraries or virtualization mechanisms, the second via efficient deterministic execution as described later.

## 3.3 Life Without Timing Channels

Our architecture requires that the provider manage scheduling and load-balancing decisions within a cloud, since enabling customers to do so would involve leaking potentially sensitive timing information into customer computations and their outputs. An important concern is whether the unavailability of this fine-grained internal timing information will make it difficult for customers to develop and optimize their parallel applications effectively: e.g., to perform detailed profiling-based analysis of their applications, or to implement application-specific dynamic optimizations or load-balancing schemes within their applications.

The unavailability of fine-grained timing information to customers may indeed present a challenge for application profiling purposes. A customer's application need not run *always* or *only* on a shared cloud, however. The customer might perform development and testing on a smaller private cloud owned or exclusively leased by the customer. Even after deployment, the customer might distribute an application across both shared and customer-private infrastructure, giving the customer access to full timing information on the physical machines the customer owns or has leased exclusively.

Some applications may require dynamic, application-specific internal load-balancing algorithms in order to perform well. To support such applications, a provider might allow customers to supply application-specific scheduling or load-balancing "plug-ins," as long as the provider's OS ensures that these plug-ins can affect *only* the application's performance and not its job outputs. The provider's OS might enforce such constraints on load-balancing plug-ins via sandboxing mechanisms for untrusted kernel extensions [7], or by running the application's load-balancing code in user space and using DIFC techniques [17, 38] to track processes that have been "tainted" with timing information, and prevent this timing information from leaking back to the customer.

# 4. A DETERMINISTIC CLOUD OS

Our architecture's "magic ingredient," obviously, is provider-enforced deterministic execution. Most cloud-oriented operating systems and virtual machine monitors replicate the inherently nondeterministic execution model provided by the underlying multiprocessor/multicore hardware. Recent application-level deterministic scheduling techniques show promise [5, 6], but they apply only within a process and do not prevent a guest from intentionally escaping its "deterministic sandbox." The only system we are aware of that enforces determinism on multiprocessor guests does so by recording and replaying a previous (nondeterministic) execution, and imposes a high performance cost [15].

To offer evidence that the proposed architecture may be practical, we introduce Determinator, a novel OS that enforces determinism on multi-process parallel computations at moderate cost, while supporting familiar parallel programming abstractions such as fork/join synchronization, shared memory, and file systems. We describe Determinator from a more general perspective elsewhere [4], but we briefly summarize here the aspects relevant to timing channel control in the cloud.

Determinator is intended to supervise the compute nodes in a cloud architecture such as that shown in Figure 2. We believe cloud providers will have an incentive to deploy deterministic compute clouds based on an OS designed along the lines of Determinator, because of the enhanced data privacy assurance that a deterministic cloud could offer security-conscious customers. Integrating Determinator into a trusted cloud computing model [31] could further increase both real and perceived security.

Our current priority is to demonstrate the viability of OS-enforced deterministic execution of compute-bound jobs. Determinator currently provides no persistent storage, and does not emulate hardware interfaces or host existing operating systems, although we intend to expand Determinator's capabilities in the future.

We now outline Determinator's basic execution environment and API, the consistency model it uses to manage state logically shared among parallel processes, and how it supports both threads inter-

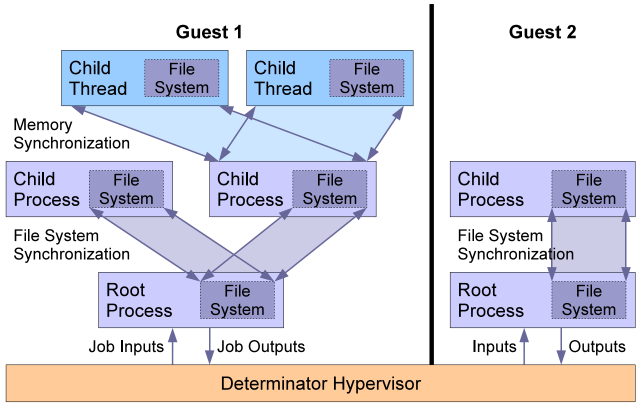


**Figure 3: Determinator process model. Each guest owns a hierarchy of processes/threads executing in parallel.**

acting via (logically) shared memory and Unix-like processes interacting via a (logically) shared file system. We make no claim that this is the "right" way to implement a determinism-enforcing OS, but merely use Determinator to explore some key design challenges and solutions, and how Determinator's design potentially addresses the goal of timing-hardened cloud computing.

## 4.1 Process Model

Determinator gives each guest an independent process hierarchy, as shown in Figure 3: it creates a *root process* on behalf of the customer, and existing processes can create new child processes. Unlike Unix, but as in nested process models [18], Determinator's hierarchy strictly constrains process lifetime and inter-process communication. A process cannot outlive its parent, and a process can communicate directly *only* with its immediate parent and children.

Although all guest processes can execute in parallel, Determinator enforces determinism in two ways. First, from the kernel's perspective, each process is single-threaded and shares *no* state with other processes. Each process has its own registers and address space, and processes cannot share read/write access to the same physical memory, thereby ensuring that each process's internal execution is deterministic as long as the processor's underlying instruction set is deterministic. Second, Determinator constrains the inter-process communication and synchronization of all processes to act as a Kahn process network [19], which provably yields deterministic behavior globally in spite of parallel execution.

## 4.2 Process Execution and API

Determinator processes can have three states: *runnable*, *stopped*, and *waiting*. Runnable processes can execute concurrently with all other runnable processes, according to a kernel-controlled scheduling policy, but do not interact with each other while running. (Processes could offer the kernel "scheduling hints" such as priorities, which the OS might use or ignore, but determinism precludes any explicit feedback from the OS affecting computed results.) A stopped process does nothing until its parent explicitly *starts* it. A waiting process is blocked until a particular child stops, at which point the waiting process becomes runnable again.

All inter-process interaction is driven by processor traps and the kernel's three system calls: PUT, GET, and RET. PUT waits until a designated child stops, then copies a block of virtual memory and/or register state into the child, and also optionally: (a) copies the child's entire virtual address space into a *reference snapshot* associated with the child; and/or (b) (re-)starts the child. GET waits until a designated child stops, then copies or *merges* a block of the child's virtual memory, and/or the child's final register state, back into the parent. A merge is like a copy, except Determinator copies only words that *differ* between the child's current and reference snapshots into the parent's address space, leaving all other words in the parent untouched. RET explicitly stops the current process, effectively returning control to the parent. Exceptions such as divide-by-zero in any process have the effect of a RET, providing the parent a status code indicating why the child stopped.

The above interaction model ensures global determinism because processes interact only at well-defined execution points determined by each process's internal flow: namely when the parent does a GET or PUT and the designated child has stopped. The kernel gives ordinary processes no ability to wait for "the first child that stops," nor to race each other to insert or remove items from message queues shared among multiple threads. See the underlying formal model [19] for more details.

If any process contains a bug causing an endless loop, other processes trying to synchronize with it might block forever. To address this risk and facilitate debugging, a processes can specify an *instruction limit* when it starts a child: the child and its descendants collectively execute at most this many instructions before the kernel forcibly returns control to the parent. Counting instructions enables processes to regain control of errant children without violating determinism, and also allows processes to "quantize" the execution of children and implement deterministic scheduling schemes [5, 13].

## 4.3 Emulating Logically Shared State

Since the kernel permits processes to share no physical state, they can communicate only by copying data via GET and PUT. The kernel uses copy-on-write to optimize large virtual copies, and uses similar techniques to optimize merge operations, so merging a page that either the parent or the child have left unmodified requires only page-level remapping. Leveraging this efficient virtual copy primitive, the C library linked into each process implements *logical* shared state abstractions purely in user space. The C library emulates shared state by treating the guest's process hierarchy like a distributed system. Each process maintains a replica of the shared state, and processes reconcile this state at well-defined *synchronization points* during program execution, as in replicated file systems [26] and distributed shared memory (DSM) systems [11].

### *Shared File System.*

Determinator's C library currently emulates the Unix file API by reading and writing a file system image stored in the process's own virtual memory. (Files could alternatively be stored in child processes not used for execution, reducing address space usage and the danger of wild memory writes corrupting shared files.)

The C library also implements Unix's `fork`, `exec*`, and `wait*` functions, to create and execute child processes whose virtual memory is not logically shared with the parent but whose file system is shared. The `fork` function clones the parent process, including file system image, into a new child process. The `exec*` functions replace the current process, *except* for its file system image, with a new executable loaded from the file system.

The `wait*` functions not only synchronize with a child process as in Unix, but also use file versioning [26] to merge the parent's and child's file system changes. The file system implements no locking or ownership, so concurrent writes to a file cause conflicts, which the C library detects and flags. A conflict makes further file access attempts return errors, until the user resolves the conflict and explicitly clears the flag (or fixes the bug causing the conflict and reruns the job). Concurrent writes are allowed in one case, how-

ever: if all writes are append-only (`O_APPEND`), as with standard output or log files, reconciliation simply collects all appends without concern for file offsets or ordering, yielding effects analogous to those of asynchronous appends in Unix.

*Shared Memory.*
Determinator's C library also emulates shared memory parallelism, currently via a simple thread fork/join API. The `tfork` function clones the entire parent process, like `fork`, but `tjoin` not only merges file system changes but also merges the child's changes to regular process memory into the parent, using the kernel's *merge* operation described in Section 4.2. The result is a deterministic analog of release-consistent DSM [11] we refer to as *deterministic consistency*, detailed elsewhere [3]. Unlike deterministic schedulers that emulate sequential consistency by executing threads under an artificial "round-robin" schedule [5, 6, 13], deterministic consistency need not rely on speculation to achieve parallelism and never needs to re-execute code due to misspeculation. Deterministic consistency also makes the effects of parallel execution not only precisely *repeatable* but also more *predictable* to the software developer. If two threads execute the statements $x = y$ and $y = x$ concurrently, for example, under deterministic consistency the result is *always* to swap the values of $x$ and $y$, whereas under deterministic schedulers the result depends on relative code path lengths and hence on subtle program input variations. Determinator's runtime can also provide deterministic scheduling for compatibility with legacy parallel code, though this execution mode has performance and predictability costs [4].

## 4.4 Implementation

An early Determinator prototype currently runs on the 32-bit x86 architecture, and implements both the shared file system and shared memory parallel APIs described above atop the kernel's deterministic "shared-nothing" processes. The prototype has no TCP/IP networking or persistent storage as yet, and merely accepts jobs from the console. The shared file system supports only 256 files, each up to 4MB in size, reflecting the limitations of a 32-bit address space. The prototype nevertheless suggests the feasibility of providing convenient and familiar parallel programming abstractions under a regime of kernel-enforced determinism.

## 4.5 Preliminary Results

To offer some evidence that the timing-hardened cloud computing architecture proposed in this paper may be feasible and efficient at least for some workloads, we briefly evaluate the current Determinator prototype using several parallel benchmarks. We use the following benchmarks: *md5* is an "embarrassingly parallel" brute-force MD5 password cracker; *matmult* is a $1024 \times 1024$ integer matrix multiply; *qsort* is a recursive parallel quicksort on an integer array; *blackscholes* is a financial benchmark from the PARSEC suite [8]; *fft* is a parallel Fast Fourier Transform from SPLASH-2 [36]; and *lu_cont* and *lu_noncont* are LU-decomposition benchmarks also from SPLASH-2. We ran all benchmarks on a 12-core (2 sockets $\times$ 6 cores), 2.2 GHz AMD Opteron PC.

Figure 4 shows each benchmark's performance running deterministically on Determinator, normalized to nondeterministic execution performance on Ubuntu Linux 9.10, using 1–12 CPU cores. Coarse-grained parallel benchmarks such as *md5*, *matmult*, and *qsort*, which perform a substantial amount of computation between inter-thread synchronization events, consistently run nearly as fast and sometimes faster on Determinator compared with Linux. The *md5* benchmark surprisingly scales much better on Determinator than on Linux, achieving more than $2\times$ speedup over Linux on 12

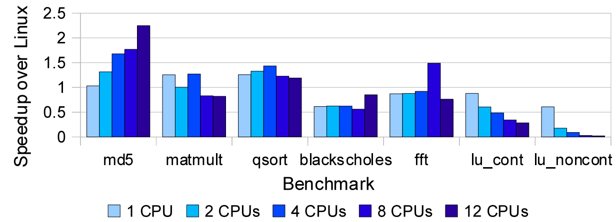


**Figure 4: Performance of several parallel benchmarks running deterministically on Determinator, versus nondeterministic execution on Linux.**

cores; we have not yet determined the precise cause of this performance increase but suspect bottlenecks in Linux's thread system [33]. The *blackscholes* benchmark is also "embarrassingly parallel," but our port of this benchmark uses deterministic scheduling for compatibility with the pthreads API, incurring a constant performance overhead [4]. The more fine-grained SPLASH-2 benchmarks exhibit higher performance costs on Determinator due to their more frequent inter-thread synchronization.

We also examined whether we could more easily reduce (though not eliminate) timing information leaks in stock Linux kernels, simply by removing access to accurate timers in both the kernel and applications. Disabling these high-resolution timers does not prevent processes from creating *ad hoc* timers via parallel threads, of course, as discussed in Section 2 and illustrated in Figure 1. Nevertheless, to test the effect of timer unavailability on a stock OS, we compiled the Linux kernel and applications to eliminate use of cycle counting instructions such as `rdtsc` and high-resolution timers. Interestingly, we found that the throughput of the Apache web server under load dropped by about 20% compared to the unmodified case, because web server and the kernel TCP/IP stack rely on high-resolution timers for estimating client latency, cache sizes, etc. This result suggests that there are no simple workarounds to close timing channels while delivering high throughput.

TCP's dependency on high-resolution timers does not present an immediate problem in our proposed cloud architecture, as long as TCP is implemented in a provider-controlled kernel or VMM: the provider's kernel is trusted and can use high-resolution timers. Dependencies on high-resolution timers in application-level suites such as Web services, however, are likely to present a pragmatic challenge when run under any timing channel control mechanism; we leave further evaluation of these challenges to future work.

# 5. RELATED WORK

Timing channels are well-studied [20, 37], but only recently examined in the cloud context [12, 30]. Most proposed solutions to recent cache-based attacks [1, 2, 28, 35] involve cache partitioning [21], requiring hardware modifications and decreasing performance. Specific algorithms may be hardened [34], but the only known general solution—resource partitioning—limits statistical multiplexing and undermines the cloud business model.

Deterministic execution has been used for other purposes such as replay debugging [22] and intrusion analysis [14], and its benefits for parallel programming are well-recognized [9, 23]. Parallel languages such as SHIM [16] and DPJ [9] provide deterministic programming models for these reasons, but they cannot run legacy or multi-process parallel code. User-level deterministic schedulers [5, 6] can provide determinism within one well-behaved process, but cannot supervise multiple interacting processes or prevent misbehaved applications from escaping the deterministic environment.

Cloud providers must be able to *enforce* determinism in guests in order to eliminate timing channels using our architecture. The only system we know of that can enforce determinism on multiprocessor guests is SMP-ReVirt [15]. While impressive, SMP-ReVirt is designed to replay prior nondeterministic executions, rather than to execute guests deterministically “from the start,” and its performance cost is too high for everyday use.

# 6. CONCLUSION

We have proposed a new, general approach to combating timing channels in clouds via provider-enforced deterministic execution. The key benefit of this approach is that it eliminates the exploitability of *all* timing channels internal to a cloud, independent of the type of resource manifesting the channel, without undermining the cloud’s elasticity through resource partitioning. Preliminary results from our determinism-enforcing OS suggest that such a timing-hardened architecture may be feasible and efficient at least for some applications, but many questions remain. Can such an architecture support fine-grained parallel applications, interactive Web applications, transactional storage- or communication-intensive applications? Can it offer cloud customers a rich and convenient, yet efficient, programming model in which to express such applications deterministically? Can deterministic clouds reuse legacy software and operating systems? Only further exploration will tell.

# 7. REFERENCES


[1] O. Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *CCAW*, Nov. 2007.

[2] O. Acıiçmez, Çetin Kaya Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, Feb. 2007.

[3] A. Aviram and B. Ford. Deterministic consistency: A programming model for shared memory parallelism, Feb. 2010. `http://arxiv.org/abs/0912.0926`.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *9th OSDI*, Oct. 2010. To appear. Preprint available at: `http://arxiv.org/abs/1005.3450`.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, Mar. 2010.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, Oct. 2009.

[7] B. N. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *15th SOSP*, 1995.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[9] R. L. Bocchino Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *1st HotPar*. Mar. 2009.

[10] D. Brumley and D. Boneh. Remote timing attacks are practical. In *12th USENIX Security Symposium*, Aug. 2003.

[11] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *13th SOSP*, Oct. 1991.

[12] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: a reality today, a challenge tomorrow. In *IEEE Symposium on Security and Privacy*, May 2010.

[13] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *14th ASPLOS*, Mar. 2009.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th OSDI*, Dec. 2002.

[15] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay for multiprocessor virtual machines. In *VEE*, Mar. 2008.

[16] S. A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *DATE*, Mar. 2008.

[17] P. Efstathopoulos et al. Labels and event processes in the Asbestos operating system. In *20th SOSP*, Oct. 2005.

[18] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.

[19] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475. 1974.

[20] R. A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *TOCS*, 1(3):256–277, Aug. 1983.

[21] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *1st CSAW*, Oct. 2008.

[22] T. J. Leblanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.

[23] E. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[24] L. Liu, E. Yu, and J. Mylopoulos. Analyzing security requirements as relationships among strategic actors. In *SREIS'022nd Symposium on Requirements Engineering for Information Security*, oct 2002.

[25] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *13th ASPLOS*, pages 329–339, Mar. 2008.

[26] D. S. Parker, Jr. et al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[27] S. Pearson. Taking account of privacy when designing cloud computing services. In *ICSE-Cloud '09*, pages 44–52, May 2009.

[28] C. Percival. Cache missing for fun and profit. In *BSDCan*, May 2005.

[29] N. R. Potlapally et al. Satisfiability-based framework for enabling side-channel attacks on cryptographic software. In *DATE*, Mar. 2006.

[30] T. Ristenpart et al. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *16th CCS*, pages 199–212. 2009.

[31] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *HotCloud*, June 2009.

[32] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *EuroSys*, Mar. 2007.

[33] R. von Behren et al. Capriccio: Scalable threads for internet services. In *SOSP'03*.

[34] C. Vuillaume and K. Okeya. Flexible exponentiation with resistance to side channel attacks. In *4th ACNS*, pages 268–283, June 2006.

[35] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *22nd ACSAC*, Dec. 2006.

[36] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd ISCA*, pages 24–36, June 1995.

[37] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, May 1991.

[38] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *7th OSDI*, Nov. 2006.